Temporal IRAD Preliminary Design and Findings

Part Two: Temporal Prototype Preliminary Design

2 July 1995

Intergraph Corporation

For questions and comments, please contact:

Gail Langran Kucera
604-360-2655
kucera@islandnet.com

This report describes the work to date on Intergraph's Internal R&D (IRAD) project on spatiotemporal modeling for geographic decision support, commonly referred to as the "Temporal IRAD." The work is described in two parts, released as separate reports. Part One (released on 4 May 95) documents the reasoning behind the many choices that have led to a provisional design. Part Two (this report) describes the provisional design itself and the prototyping plans. A final report will be released upon completion of the prototype.

Gail Kucera, the IRAD's principle investigator, authored this report. However, the thoughts of many are represented here through literature review and lengthy discussions internal to Intergraph. In particular, Harold McDaniel, Janet Conklin, and Sam Bacharach were generous with their expertise and advice, which was gratefully received and incorporated within.

Temporal IRAD Preliminary Design and Findings

Part Two: Provisional prototype design

This report describes the logical model of the Temporal IRAD prototype. Because the prototyping process is considered to be research, not development, this design is the starting point upon which we will build. For background on the rationales for design elements described here, refer to Part One: Design Choices and Rationales, issued by Intergraph in early May 1995.

Spatiotemporal modeling is of interest to all GIS applications, although to varying degrees. At minimum, modeling database changes through linked feature versions is a natural way to manage transactions and facilitate database rollbacks. This permits us to remove all changes made from a faulty data source, by a particular data analyst, or in a particular timeframe (for example, during a period of system crashes). It also fosters accountability among data users, because we can pose the question, "What did we know and when did we know it?" Modeling changes in the world is a second thread of temporality; the discussion that follows distinguishes the two at length. Suffice to say that the history of features as they changed in the world (assuming that the change is asynchronous to database changes) permits us to ask questions of causality and attempt to extrapolate into the future.

The data modeling described in this report is intended to address the most difficult temporal requirements, and therefore to form a superset of what would be a temporal solution for most systems. The requirements addressed include a high degree of uncertainty, asynchronicity of world and database, a need for prediction, problems with error, a desire to discriminate individual attribute changes, and rapid performance. The sections that follow present a basic object model, discuss critical aspects and tradeoffs of temporal data management, describe how various situations would be handled within the object model, and add detail progressively.

1. Object model

The intent of this work is to model geographic features as they change over time. A feature is characterized by its behavior and location in time and space. Feature character is translated to a storage object via defined attributes and geometry.

The object model for temporal features has four classes: feature, version, geometry, and attribute. Each feature has one or more versions, and each version has a geometric description and one or more attributes. Figure 1 describes the object model using OMT (Object Modeling Technique) notation. A box represents a class. The box's middle part is reserved for class attributes; its lower part for class methods. Note that physical objects in the intended prototype implementation may not correspond exactly with the logical objects described here. The object model emphasizes conceptual clarity; the object implementation will emphasize performance.

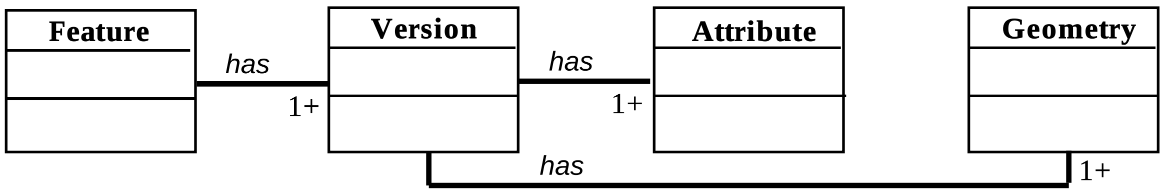


Figure 1. The object model for the prototype temporal system. Each feature has one or more versions, and each version has one or more attributes and geometric elements.

2. World time vs. database time

Two classes of time are clocked in the temporal prototype: world time and database time. World time is when feature data were collected in the world (DTOI for imagery) or (if available) when a feature change actually occurred. Database time is when the data were committed to the database.

The possible relationships between world and database time should be constrained to the extent possible for integrity control. Most systems can safely constrain the relationship so world time always precedes database time (Figure 2a). Thus, a change occurs in the world, it is observed, then it is recorded in the database, in that sequence. It is possible for database time to precede world time if a change is planned or expected, then recorded in the database with a future effective date in world time (Figure 2b). Both situations are possible in TFG (Figure 2c); for that reason, the temporal prototype will not constrain the relationships between world and database time.

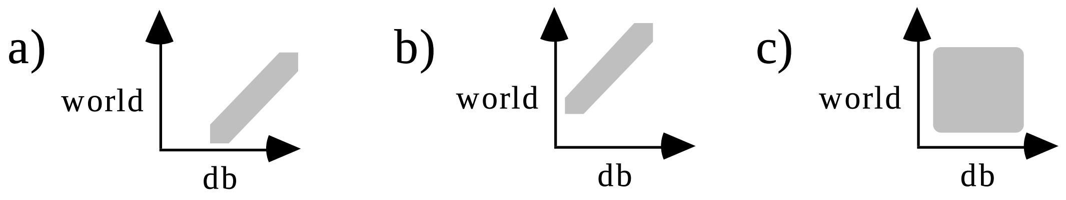


Figure 2. Possible relationships between world and database time. a) World time leads database time when changes observed in the world are later recorded in the database. b) World time lags database time when changes are recorded in the database before they occur in the world. c) Systems such as TFG can support both relationships between world and database time.

Timestamp relationships are rich sources of temporal information (Table 1). They tell how timely the information is, the extent of future plans, rate of change or rate of data collection, rate of database updates, and orderliness of the updating process.

Table 1. The meanings of relationships between timestamps for a given feature version.

World time before db time	The difference is the ***timeliness*** of the information.
DB time before world time	The difference is the degree of ***extrapolation*** into the future.
World times between versions	The difference is the ***rate of change***, combined with the ***amount of intelligence*** available in that area.
DB times between versions	The difference is the ***rate of data capture*** during that period. During TFG's cold start, the intervals between db times are likely to be very short.
World and db times between versions	The less linear the correlation, the more retroactive, expected, or corrective information has been inserted.

Because timestamps are so useful, the temporal prototype will use them liberally. This provides an upwardly mobile system that is capable of rationally describing complex sequences of events that change the status of features. The prototype will not, however, include code to exploit all the descriptive properties of timestamps. These can be added judiciously as requirements are explored further.

3. Bracketing of effective dates

Each feature "lives" for a given period in the world and in the database. Likewise, each version has a period of effectiveness, as does each component of its attribute and geometric description (Figure 3). A period of effectiveness is a timespan bracketed by start and end times.

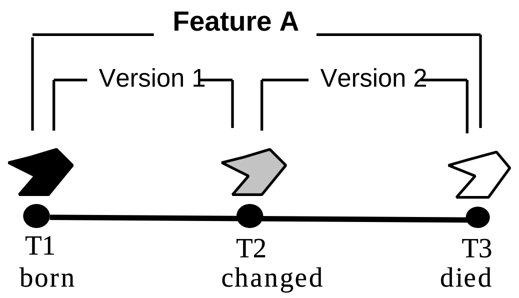
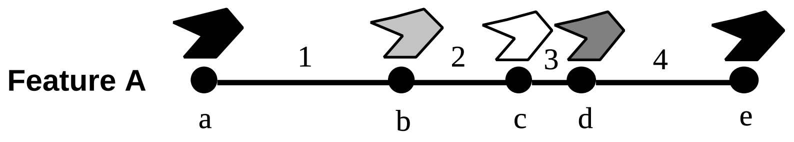


Figure 3. Bracketing of lifespan and periods of effectiveness. Feature A was "alive" or "effective" from T1 to T3. Version 1 was effective from T1 to T2.

In a system with a linear temporal topology (i.e., where only one version of a given feature is effective at any given moment), it is possible to allow end time of versions and attributes to be implicit if versions can be sorted sequentially by effective dates (Figure 4a). Alternately, one can explicitly include a timestamp to close the temporal interval when a version (and by extension, an attribute) is effective (Figure 4b). Table 2 presents the pros and cons of explicit bracketing. In OM, we have the additional option of maintaining the versions of a feature in sorted order of database time using relatively ordered channels.



a) implicit
1: from node a
2: from node b
3: from node c
4: from node d

b) explicit
version: from node a to node
version: from node b to node
version: from node c to node
version: from node d to node

Figure 4. Bracketing of effective dates for versions. a) If versions are sorted, the end of an interval implicitly coincides with the beginning of the previous version. b) If effective dates are bracketed at start and end, versions need not be sorted and complex temporal topologies are supported.

Table 2. Pros and cons of implicit vs. explicit bracketing of effective interval.

Implicit			**Explicit**
Reduces storage via fewer timestamps	❖		Additional timestamps
Only the simplest (linear) temporal topology is supported		❖	Branching and complex temporal topologies are supported
Error correction options are limited		❖	More options for error correction, uncertainty
To find ***end date***, must find next version		❖	Entity described completely by its attributes

The temporal prototype will explicitly bracket effective intervals for features and versions. Feature birth and death will be bracketed in both world and database time, to permit rapid access to features existing at a moment. Versions will be bracketed in world and database time to permit rapid access to information about the current feature state at a moment.

Attributes and geometry also have periods of effectiveness. We plan to timestamp attributes and geometry only with effective start time in world time. End time and database times can be derived by traversing the temporal data structure, should this information be required. Table 3 and Figure 4 summarize how timestamps are used for each logical entity in the object model. Figure 5 shows how they bracket features and versions. Figure 6 illustrates the mechanics of feature, version, and attribute timestamping.

Table 3. Timestamps planned for the temporal prototype.

Entity	Timestamp	Primary query supported
Feature	world birth	When was the earliest sighting of this feature?
	world death	When did the feature cease to exist?
	db birth	When did the feature first enter the db?
	db death	When was the feature registered as nonexistent?
Version	world start	What was the effective state of the feature at this time?
	until date	When was the last time the state was confirmed by source?
	world end	When was the world state superseded by another?
	db start	What did the db show to be effective at this time?
	db end	When was the db state superseded by another?
Attribute	world start	Since when did this attribute hold this value?

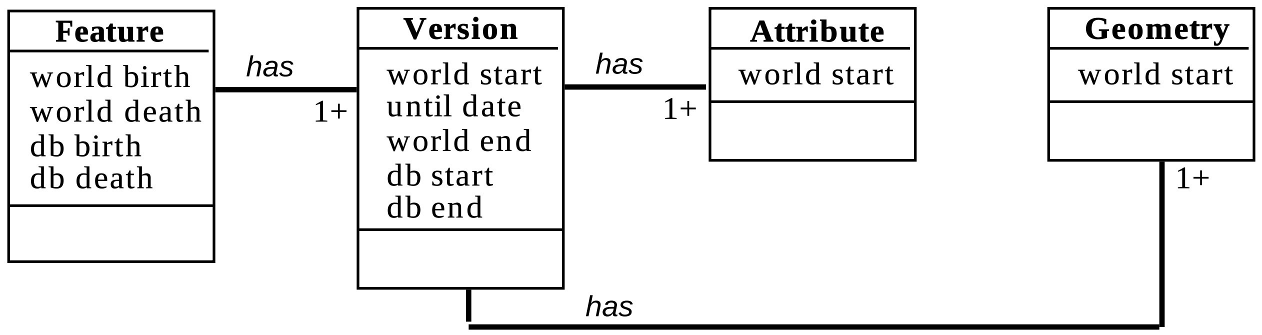
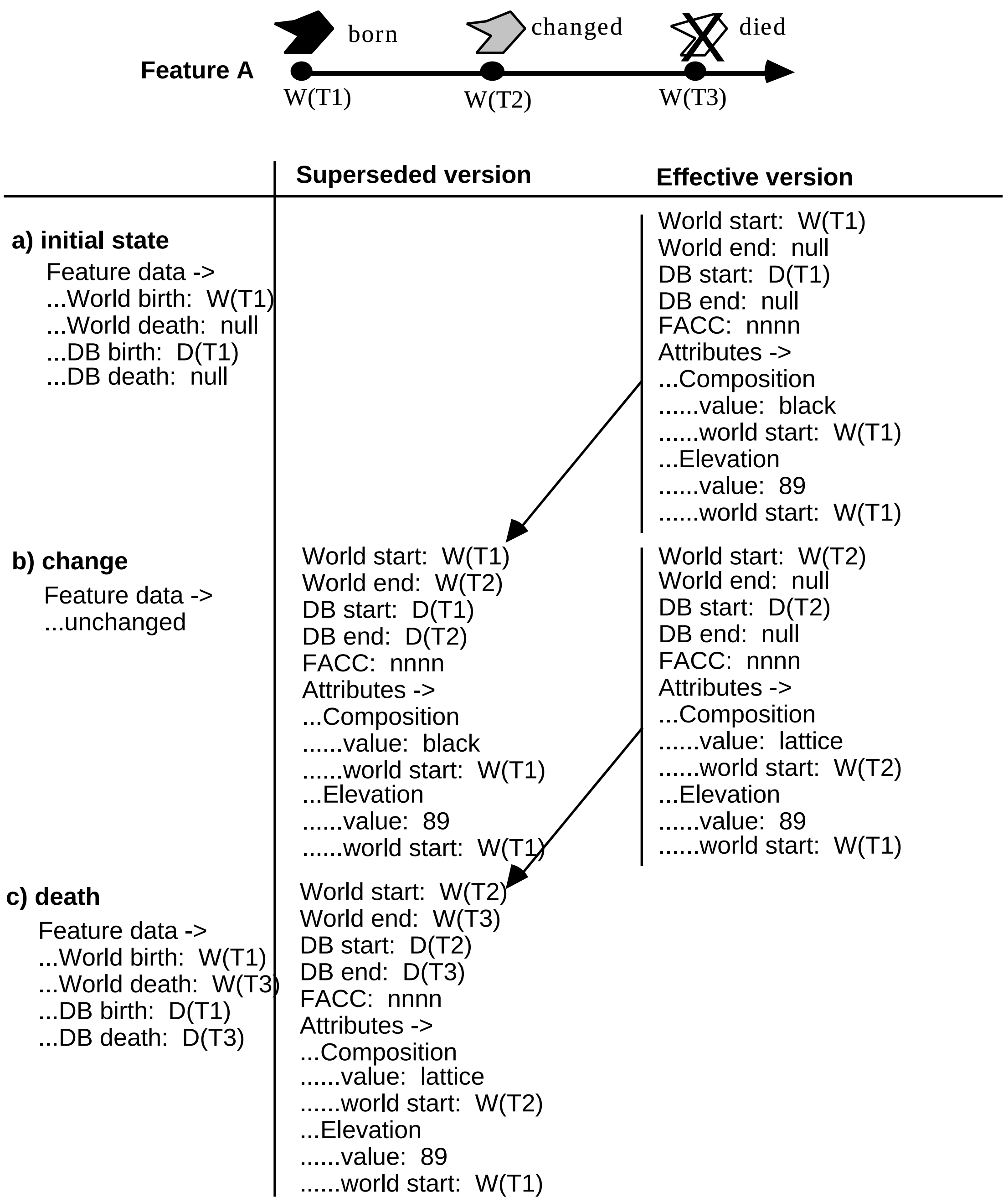


Figure 4. Object model with temporal attributes added.



	Superseded version	Effective version
a) initial state Feature data -> ...World birth: W(T1) ...World death: null ...DB birth: D(T1) ...DB death: null		World start: W(T1) World end: null DB start: D(T1) DB end: null FACC: nnnn Attributes -> ...Composition ......value: black ......world start: W(T1) ...Elevation ......value: 89 ......world start: W(T1)
b) change Feature data -> ...unchanged	World start: W(T1) World end: W(T2) DB start: D(T1) DB end: D(T2) FACC: nnnn Attributes -> ...Composition ......value: black ......world start: W(T1) ...Elevation ......value: 89 ......world start: W(T1)	World start: W(T2) World end: null DB start: D(T2) DB end: null FACC: nnnn Attributes -> ...Composition ......value: lattice ......world start: W(T2) ...Elevation ......value: 89 ......world start: W(T1)
c) death Feature data -> ...World birth: W(T1) ...World death: W(T3) ...DB birth: D(T1) ...DB death: D(T3)	World start: W(T2) World end: W(T3) DB start: D(T2) DB end: D(T3) FACC: nnnn Attributes -> ...Composition ......value: lattice ......world start: W(T2) ...Elevation ......value: 89 ......world start: W(T1)	

Figure 6. Prototype plan for bracketing effective dates in world and db time. Attributes are bracketed explicitly in world time. Versions are bracketed explicitly in db time.

4. The role of timestamps in information retrieval

The timestamps in the prototype system provide the basis for responding to a wide variety of temporal queries. Table 4 lists some logical constructs that can be created through use of timestamps. Table 5 describes typical queries and their means of response.

Table 4. Logical constructs using timestamps. TOI(W) and TOI(D) are "time of interest" in world and db.

Existing in world	MAX[Where Feature.world birth < TOI(W)< Feature.world death][1]
Existing in db	Feature.db death < TOI(D) < Feature.db death
Lifespan (W)	Feature.world death - Feature.world birth
Lifespan (D)	Feature.db death - Feature.db birth
State (W)	Version.world start < TOI(W)< Version.world end OR [Version.world start < TOI(W) AND Version.world end = NULL]
State (D)	Version.db start < TOI(D)< Version.db end OR [Version.db start < TOI(D) AND Version.db end = NULL]
State duration (W)	Version.world end - Version.world start
State duration (D)	Version.db end - Version.db start
Attribute volatility	TOI(W) - Attribute.world start

Table 5. Sample queries using timestamps.

To retrieve a snapshot at a TOI(W) in an AOI:

```
For all features within the Area(AOI),
    For all features Existing in world at Time(TOI)
        Find Feature state(W) at Time(TOI)
```

To retrieve a snapshot showing what was known about the AOI as of TOI(D):

```
For all features within the AOI,
    For all features Existing in db at TOI(D)]
        Find Feature state(D) at TOI(D)
```

To find the history of a feature of interest: Sort all versions of the feature of interest by world birth

To find transactions to a feature of interest: Sort all versions of the feature of interest by db birth

To find the last world change before a TOI(W) to a feature of interest:
Of the versions where world end < TOI(W), find the one with the highest world end value

To find the next world change after a TOI(W) to a feature of interest:
Of the versions where world start > TOI(W), find the one with the lowest world start value

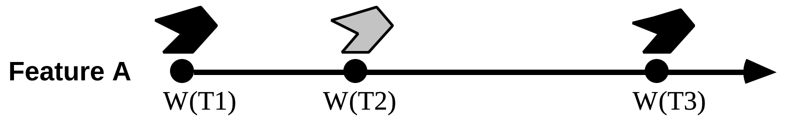
To find the last db transaction before a TOI(D) to a feature of interest:
Of the versions where db end < TOI(D), find the one with the highest db end value

To find which attributes changed to cause a new feature version:
Find the attributes where attribute.world start = version.world start

5. Expressing temporary changes

[1]There may be duplicate versions at one timeslice for error correction. Such "ties" in time are broken by looking to database time.

Some changes are temporary. The temporal prototype will automatically reverse a temporary change when its period of effectiveness ends (e.g., when *current time = end time)*. One approach is to embed a trigger in the *end time* attribute (Figure 7).



	Superseded version	**Effective version**
a) initial state		World start: W(T1) World end: null DB start: D(T1) DB end: null FACC: nnnn Attributes -> ...Composition ......value: black ......world start: W(T1) ...Elevation ......value: 89 ......world start: W(T1)
b) temporary change	World start: W(T1) World end: W(T2) DB start: D(T1) DB end: D(T2) FACC: nnnn Attributes -> ...Composition ......value: black ......world start: W(T1) ...Elevation ......value: 89 ......world start: W(T1)	World start: W(T2) World end: null DB start: D(T2) DB end: null FACC: nnnn Attributes -> ...Composition ......value: black ......world start: W(T2) ......(a trigger will reverse this ...... change as specified) ...Elevation ......value: 89 ......world start: W(T1)
c) revert to original	World start: W(T2) World end: W(T3) DB start: D(T2) DB end: D(T3) FACC: nnnn Attributes -> ...Composition ......value: black ......world start: W(T2) ......(trigger fired) ...Elevation ......value: 89 ......world start: W(T1)	World start: W(T3) World end: null DB start: D(T3) DB end: null FACC: nnnn Attributes -> ...Composition ......value: black ......world start: W(T3) ......(trigger fired) ...Elevation ......value: 89 ......world start: W(T1)

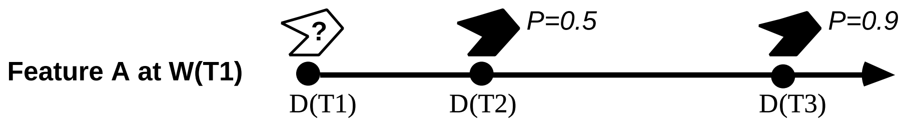
Figure 7. Treatment of temporary changes. a) Original version before temporary change. b) A temporary change is entered. c) The trigger reverses the change by creating a new version.

6. Confirming inferred information

Information on a given feature may come in pieces. Evidence of its existence may be gleaned before details of its attributes are available; alternately, attributes may be added with low certainty levels, then later confirmed. Two approaches are possible when information arrives incrementally. One is to treat each stage of loading as a distinct version of the feature. Another is to version only when a *change* has occurred, not when more information is added or existing information is confirmed. Table 6 lists pros and cons of the two approaches. The main advantage of not versioning is that fewer versions result. However, this treatment is inconsistent with the treatment of other changes and updates to the database. For that reason, we will version features when any unfilled attribute is filled, or when a certainty value for an attribute is changed. Figure 8 shows our provisional approach.

Table 6. Pros and cons of versioning on confirmed or additional information.

Versioning			No versioning
Shows when each item of information was available	❖		Lesser accountability within the database
Treats additional information and improved certainty exactly as changes and corrections are treated	❖		Is not consistent with treatment of other new information
Makes no assumptions about newly available information	❖		Assumes newly available information was true previously, if no prior information exists
Wasteful of storage due to additional versions		❖	Thrifty use of storage
Possibility of slower retrievals due to more versions		❖	Fewer versions could mean faster retrievals
Best if retroactive judging of decision-making may occur	-	-	Best if all critical uses of the database are in the present tense



	Superseded version	Effective version
a) initial discovery		World start: W(T1) World end: null DB start: D(T1) DB end: null FACC: nnnn Attributes -> ...Composition ......value: null ......world start: null ......certainty: null ...Elevation ......value: null ......world start: null ......certainty: null
b) fill an attribute	World start: W(T1) World end: null DB start: D(T1) DB end: D(T2) FACC: nnnn Attributes -> ...Composition ......value: null ......world start: null ......certainty: null ...Elevation ......value: null ......world start: null ......certainty: null	World start: W(T1) World end: null DB start: D(T2) DB end: null FACC: nnnn Attributes -> ...Composition ......value: black ......world start: W(T1) ......certainty: 0.5 ...Elevation ......value: null ......world start: null ......certainty: null
c) improve certainty	World start: W(T1) World end: null DB start: D(T2) DB end: D(T3) FACC: nnnn Attributes -> ...Composition ......value: black ......world start: W(T1) ......certainty: 0.5 ...Elevation ......value: null ......world start: null ......certainty: null	World start: W(T1) World end: null DB start: D(T3) DB end: null FACC: nnnn Attributes -> ...Composition ......value: black ......world start: W(T1) ......certainty: 0.9 ...Elevation ......value: null ......world start: null ......certainty: null

Figure 8. Treatment of inferred and added information. Three successive sources of the same date are available at three different times. a) Original discovery of feature without attribute values. Note that world time also is stored at the feature level to describe date of birth. b) The composition value is added with a high degree of uncertainty. c) The certainty of the composition value improves.

7. Error correction

Many of the complexities of temporal systems are due to ambitious error-correction requirements. There will be no shortage of errors in TFG; the goal is to decide in advance which errors are important to correct and which can be tolerated. Imagine the sequence where a new feature enters the database, new source from an earlier date shows the feature in a different light, and later source shows the original source was erroneous. Figure 9 illustrates some of the error cases that could occur in TFG, since source will arrive and be applied out of sequence, and different source documents could provide conflicting information about features. Note that error can occur in initial birth date, in the past tense, and in the present tense. Table 7 illustrates the complexity that would occur if all such errors were recorded and managed via versioning.

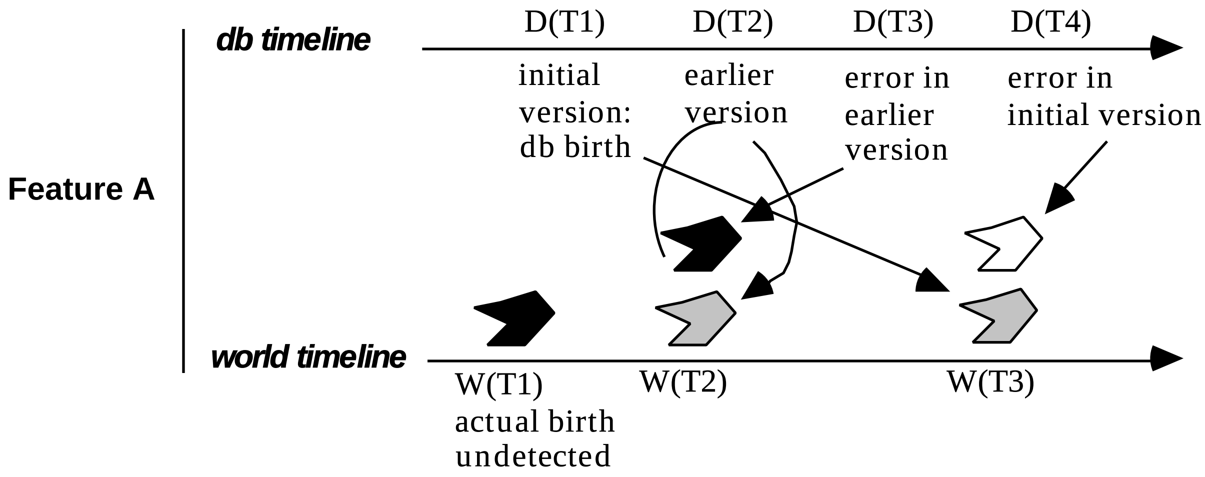


Figure 9. Discovery of error. The example shows a feature being born in the database that existed for some time previously in the world. Errors are found in the present and past.

One of the early motivations for research in temporal databases was to improve on transaction-rollback methods of viewing past database states. Today, a crucial element of temporal system design is to determine a minimum set of error correction measures to avoid needless complexity. Table 8 enumerates potential error correction requirements, approaches, and the queries they support. Of the three requirements, each is successively more difficult to achieve. Within a requirement, the approaches are grouped from lowest to highest impact on system cost and complexity. Thus, the final entry in the table (to correct the past and to explain the error) adds considerable cost and complexity to a system and should be attempted only with strong justification.

Table 7. Error traffic caused by out-of-sequence and poor-quality source. If every error were addressed using versioning procedures, versions would be interleaved in world time as shown. The table describes the situation expressed by the source, the relationship between world and database time, and the new sorting of versions. Changes are in bold underlined type.

Situation	Versions sorted by world time
Source 1 shows a new feature (W<D)	**V1**
Source 2 shows the same feature at an earlier time with different attributes (W<D)	**V2** V1
Source 3 shows the same feature at an even earlier time with the same attributes (W<D)	**V2/V3*** V1
Source 4 shows a distant future change (W>D)	V2/V3 V1 **V4**
Source 5 shows an immediate change of limited duration (W<D)	V2/V3 V1 **V5** V4
Source 5's duration expires (W = D)	V2/V3 V1 V5 **V6** V4
Source 4's expected change is canceled (W<D)	V2/V3 V1 V5 V6 **V4/V7***
Source 1 was erroneous (W<D)	V2/V3 **V1/V8*** V5 V6 V4/V7
Source 5's temporary change was erroneous	V2/V3 V1/V8 **V5/V9*** V6 V4/V7

The simplest way to correct errors is to replace the offending information with the correct information. However, replacement prevents the user from reviewing the past and finding the temporary presence of an error. Such information can be key, particularly when judging the goodness of decisions and the performance of decision-makers. To do this, the database must describe what information was available to the decisionmaker at the time of the decision, even after subsequent updates and corrections. A better approach to error correction is to superseded the incorrect information with correct information, clocking changes in database time. This strategy also provides a graceful means of database rollback.

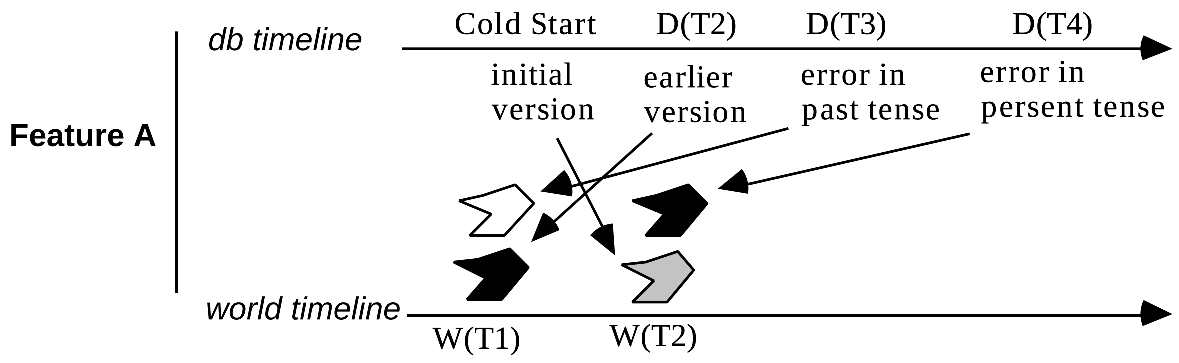
The model for update and error correction in a temporal system is a transaction-based financial accounting system. By making a later entry to correct an earlier error, one supersedes but never deletes information. Information is additive; thus, the accountant posts entries when a payment comes due, payment is received, check bounces, and a new payment is received. If a check is entered erroneously, the error is recorded as a transaction; the original (erroneous) entry is not erased or overwritten. The system provides the answer to the questions: what did we know? And when did we know it? In the case of the accounting system, the goal is to know what accounts are payable, received, and cash on hand. How to handle geographic transactions relates directly to what information the system must provide.

Table 8 summarizes options for feature update, present-tense error correction, and correction of errors in data that already are superseded. Its purpose is to assist a prospective user to carefully assess the importance of gaining responses to the queries listed in Column 2. It should be clear from Table 7 that retroactive error correction comes at great cost, and should only be undertaken with clear and compelling requirements.

Table 8. Error correction requirements and approaches, ranked in impact on system cost and complexity.

Requirement/Approach	**Queries supported**
Requirement: update feature data Approach:	
- Overwrite outdated info	What is the state of this feature?
- Supersede outdated info keeping a historical record of changes (W)	How has this feature changed over time?
- Supersede outdated info via transaction log (D)	How did our db describe this feature over time?
- Supersede outdated info, maintaining a history (W) and transaction log (D)	How has this feature changed, and how correct was our db? What was the time lag?
Requirement: correct present-tense data Approach:	
- Overwrite incorrect info	What is the state of this feature?
- Supersede incorrect info with correction (W), maintaining transaction log (D)	How did our db describe this feature over time?
- Supersede incorrect info with correct ion (W), plus info that describes error detection (W) and transaction log (D)	What are the competing certainties of the correct and (probably) incorrect info? How and when was the error detected?
Requirement: correct past-tense data Approach:	
- Don't	What did we know in the past?
- Insert corrected info (W), maintaining transaction log (D)	How does what we knew compare to what we now know?
- Insert corrected info (W), plus info that describes error detection (W), plus transaction log (D)	What are the competing certainties of the correct and (probably) incorrect info in the past tense? How and when was the error detected?

Figure 10 illustrates the suggested error-correction strategy. However, no error correction will be prototyped due to time restrictions and a need to investigate requirements at a finer level of detail.



	Superseded version	Effective version
a) initial version W(T2) Feature data -> ...World birth: W(T2) ...World death: null ...DB birth: Cold Start ...DB death: null		World start: W(T2) World end: null DB start: Cold Start DB end: null FACC: nnnn Attributes -> ...Composition ......value: open lattice ......world start: W(T2) ...Elevation ......value: 89 ......world start: W(T2)
b) earlier version W(T1)	**(no action; system does not correct the past)**	
c) correction at W(T1)	**(no action; system does not correct the past)**	
d) correction at W(T2) Feature data -> ...unchanged	World start: W(T2) World end: null DB start: Cold Start DB end: D(T4) FACC: nnnn Attributes -> ...Composition ......value: open lattice ......world start: W(T2) ...Elevation ......value: 89 ......world start: W(T2)	World start: W(T2) World end: null DB start: D(T4) DB end: null FACC: nnnn Attributes -> ...Composition ......value: black ......world start: W(T2) ...Elevation ......value: 89 ......world start: W(T2)

Figure 10. Recommended error correction for TFG and for most temporal systems. Only errors in the present tense are corrected unless strong justifications for past-tense corrections are found to exist.

In sum, one must be pragmatic about what aspects of error are of interest and in what form the information would be needed. Pending detailed discussion of TFG's temporal requirements, we suggest the following strategy:

- During cold start, data capture creates a unified base state, where each feature has only one version, compiled from the best-available information, timestamped with its source dates, and posted with its certainty values. The base state is comparable to a map, where features have differing lineage and currencies, but conflicts are resolved before capture.

- Following cold start, corrections are made only on present-tense versions. Updates, too, would be made only on present-tense versions *unless clear requirements exist to "densify" the past-tense temporal data* in TFG.

8. Temporal data management methods

Methods are procedures associated with an object. Our logical model has four primary components: feature, version, attribute, and geometry. Because physical implementation will differ from the logical description presented here, we do not link methods with logical objects but describe them as they relate to the system as a whole. The temporal prototype will implement a subset of the following methods.

CREATE FEATURE
Update the database with a new feature. This method is invoked when the feature identity changes so extensively that the feature is now a new feature.
- Create a new feature structure
- Timestamp world birth
- CREATE VERSION

CREATE VERSION
Update the database to show a new present-tense state of the feature.
- Create a new version structure
- Timestamp world start
- If there is an existing version, SUPERSEDE VERSION, using the new version's world start as the superseded version's world end
- Insert new version in correct temporal sequence, if physically ordered

----

END FEATURE LIFESPAN
End the lifespan of the feature.
- Timestamp world death
- SUPERSEDE VERSION
- Commit to db; timestamp db death

SUPERSEDE VERSION
Move the outdated feature state to the past tense.
- Timestamp world end
- Move superseded graphics to vector graphics
- Commit to db; timestamp db end

FIND LATEST VERSION
Locate the present-tense version.
- If versions are kept in sorted list (relatively ordered channel), select last item
- If versions are sorted logically, find version where world end = NULL AND MAX(db start)

FIND LATEST UPDATE
Locate the most recent change to the feature.
- Find version where MAX(db start)

FIND EARLIEST VERSION
Locate the earliest manifestation of the feature.
- Find version where MIN(world start) AND MAX(db start)

FIND FIRST ENTRY
Locate the first version created for the feature.
- Find version where MIN(db start)

FIND STATE AT DATE (WT)
Locate the correct version for a given date (WT).
- Find version where world start < WT AND (world end > WT OR world end = NULL) AND MAX(db start)

FIND STATE AS OF DATE (DT)

Locate the version effective in the db as of the given date (DT).

- Find version where db start < DT AND (db end > DT OR db end = NULL)

INTEGRATE NEW FEATURE DATA

Take new feature information and process it. Only the "update" methods invoked by this method (i.e., not the error correction or certainty improvement methods) are described in this section. Others will be added if they are undertaken in the prototype.

```
• If it is a new feature, then CREATE FEATURE; exit
• If the feature has died, then END FEATURE LIFESPAN; exit
• If it is an update to a feature
      then if it includes changes to identity attributes
         then UPDATE IDENTITY ATTRIBUTES
         else if it includes changes to domain attributes
            then UPDATE DOMAIN ATTRIBUTES
            else if it includes changes to versioning attributes
               then UPDATE VERSIONING ATTRIBUTES
               else if only graphics are changed
                     then UPDATE GRAPHICS
• If it is a correction to a feature
      then if it includes changes to identity attributes
         then CORRECT IDENTITY ATTRIBUTES
         else if it includes changes to domain attributes
            then CORRECT DOMAIN ATTRIBUTES
            else if it includes changes to versioning attributes
               then CORRECT VERSIONING ATTRIBUTES
               else if only graphics are changed
                     then CORRECT GRAPHICS
• If it is an improvement to a feature (via more certain or additional attributes)
      then if it affects identity attributes
         then IMPROVE IDENTITY ATTRIBUTES
         else if it affects domain attributes
            then IMPROVE DOMAIN ATTRIBUTES
            else if it affects versioning attributes
               then IMPROVE VERSIONING ATTRIBUTES
               else if only graphics are affected
                     then IMPROVE GRAPHICS
```

UPDATE IDENTITY ATTRIBUTES

Change the feature identity, which causes the old feature to die and a new one to be born.

- CREATE FEATURE
- Populate version attributes with NULL values
- Populate graphics with existing graphic
- Insert attribute updates found in the update packet
- Insert graphic updates found in the update packet
- Commit new feature and version to db, timestamping with db birth and db start.
- END FEATURE LIFESPAN of previous feature, using the source date of the new identity attribute as the world death date.

UPDATE DOMAIN ATTRIBUTES

Integrate updates that may cause the feature to die and be reborn.

- Check all domain attributes to see if the feature will die and be reborn as another.
- If any update will cause the feature to die and be reborn

```
    then
        CREATE FEATURE
        Populate version attributes with NULL values
        Copy existing graphics to the new version
        Insert attribute updates found in the update packet
        Insert graphic updates found in the update packet
        Commit new feature and version, timestamping db birth and db start
        END FEATURE LIFESPAN of previous feature
    else
        CREATE VERSION
        Copy existing attributes to the new version
        Copy existing graphics to the new version
        Insert attribute updates found in the update packet
        Insert graphic updates found in the update packet
        Commit the new version, timestamping db start
```

UPDATE VERSIONING ATTRIBUTES

Insert attribute changes that have no effect on feature life and death, but rather cause a new feature version to be created.

- CREATE VERSION
- Copy existing attributes to the new version
- Copy existing graphics to the new version
- Insert graphic updates found in the update packet
- Commit the new version, timestamping db start

UPDATE GRAPHICS

Insert graphic changes that have no effect on feature life and death, but rather cause a new feature version to be created.

- CREATE VERSION
- Copy existing attributes to the new version
- Copy existing graphics to the new version
- Insert graphic updates found in the update packet
- Commit the new version, timestamping db start

9. Temporal integrity constraints

Table 9 lists integrity constraints that are possible in a working system. However, the temporal prototype will enforce no integrity constraints (due to time constraints).

Table 9. Possible temporal integrity constraints

For any pair of world/db timestamps within a feature or a version (e.g., world birth/db birth; world death/db death; world start/db start; world end/db end) :

> world time _ db time
> world time _ "db time + 30 (or less, as source timeliness improves)

For any set of versions of one feature

> db start (superseded version) < db start (new version)
> db end (superseded version) = db start (new version) +/- a small tolerance
> world end (superseded version) = world start (new version) +/- a small tolerance
> IF world start (version a) = world start (version b) THEN db start (version a) _ db start (version b)

10. Issues of geometry and topology

The following basic requirements exist for handling the geometric and topological descriptions of features.

- Provide fastest possible present-tense performance; do not permit past-tense data to impact speed of response to present-tense queries.

- Provide a capability to retrieve the geometric description of a feature at any moment from Cold Start to the present. Topological relationships can be recalculated rather than stored for past-tense data.

- Provide a capability to symbolize past-tense graphics.

- Provide a capability to use past-tense graphics in such analytical procedures as network analysis, buffering, and size / distance / adjacency computations.

Our approach for treating geometric and topological descriptors is to treat the present-tense data in the manner developed for TFG's use of OM: in topologically integrated datasets that use the high-speed topological operators of OM. Thus, the present tense of the temporal prototype will appear and function much as would an atemporal GIS.

When a feature in the prototype changes geometrically, its past-tense version will be moved from present-tense "topological data" to "Vector Graphics." Vector Graphics do not include topological referencing and thus carry far lower storage overhead. Vector Graphics are analyzed using a set of analytical subroutines separate from those used to analyze topological graphics. Vector Graphics analysis is somewhat slower than the topologically references analysis of topological graphics. If sustained or intensive analysis is needed, Vector Graphics can be re-integrated into a topological level.

The rationale for moving past-tense spatial data from topological to vector graphics is pragmatic.

- If all data at all times are topologically intersected, the successively smaller fragments become increasingly harder to manage.
- Vector graphics, having originated in the topology layer, are topologically clean before being moved.
- No practical reason seems to exist for topological intersection of features that existed at different times.
- By moving old versions from the present-tense data stores, we ensure that the performance of present-tense analysis does not suffer from the presence of past-tense information.

11. Measures to facilitate analysis

The requirements we aim to meet go beyond merely storing and retrieving temporal information about geographic features. The temporal features must be analyzed in a variety of ways, and be conducive to analysis at any stage of database loading. An important requirement for analysis is rapid retrieval; this will be addressed by architecture and data structuring. But a second important requirement is that data are stored in a meaningful way that preserves all available information.

The FACCS coding system provides a feature-attribute breakdown that describes basic feature information. However, additional information is needed to describe changes to features. Earlier sections in this report discuss the need for timestamps and versioning. Of equal importance is to describe *agents of change* to the extent that they are known.

Information about feature changes will enter the system in two ways. The first is through source documents that describe the state of a geographic area or feature. For our purposes, these are "state data." A second way to acquire information is through knowledge of an agent of change–a flood, snowstorm, avalanche, earthquake, or bombing–that is likely to affect feature states. For our purposes, these are "event data." The documents that describe state and event data can be considered "evidence." Thus, states, events, and evidence are three critical components of temporal information. Each must be described independently to ensure that all data are carefully preserved (Figure 11). Table 10 lists some states, events, and evidence applicable to TFG. Each should be treated as a "feature" of sorts, since each has areal extent, temporal duration, and descriptive attributes.

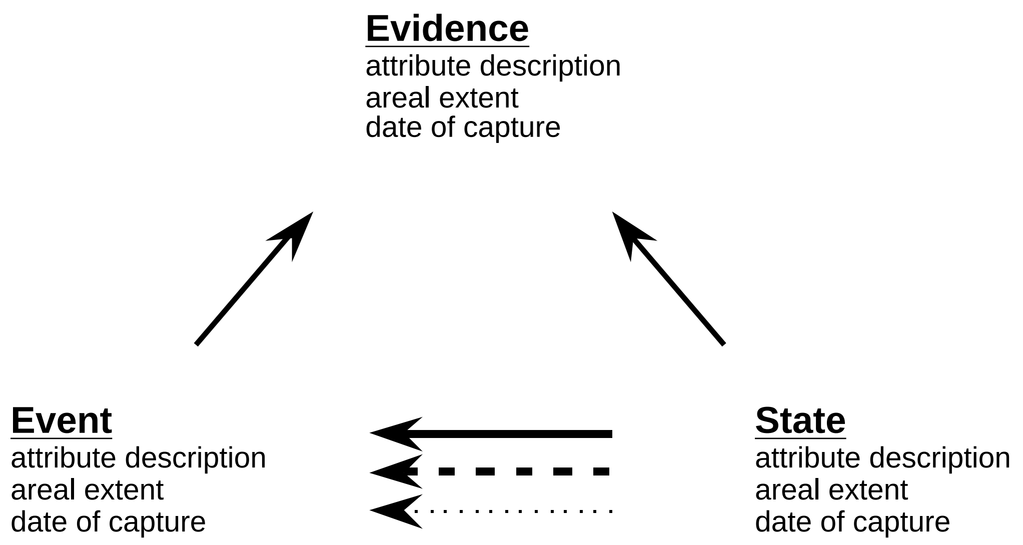


Figure 11. The interplay of states, events, and evidence. Note that states, events, and evidence all have spatial and temporal extent, plus attributes. States and events both should be referenced to the available evidence that describes them. Most important, many state changes are predicated on the events that precede them. In many cases, we can hypothesize state change following an event that overlaps its area, while awaiting new evidence on the state.

Table 10. Examples of TFG states, events, and evidence. The new "event" features should be assigned any attributes necessary to hypothesize state changes in their areas.

States	**Events**	**Evidence**
roads	bombing	ground survey
streams	heavy rainfall	air survey
bodies of water	heavy snowfall	reconnaissance reports
bridges	heavy wind	maps and charts
vegetation areas	flooding	
buildings	earthquake	
ice masses	avalanche or landslide	

One reason to disaggregate states and events is that events are known to affect states, but the precise effect is unknown until new evidence arrives to describe it. Treating an event as a special class of feature permits us to hypothesize or model change while awaiting reconnaissance data. It also permits later analysis of causality, i.e., what state changes or events historically have followed other state changes or events, and why?

Object-oriented tools permit the inclusion of triggers in event features, which can be designed to modify state features that are intersected. Thus, the system can be designed to automatically precipitate hypothesized state changes (presumably with low certainty values) upon input of event features. Automated state changes would be reversable by an analyst if they contradicted what the analyst believed to be true, but they would be designed to mimic the information the analyst would input manually in the absence of better information, and to incrementally improve the timeliness of the data while awaiting the receipt of better evidence.

To illustrate how events might trigger state changes, imagine a bombing where a road is known to have been hit. Air reconnaissance may be available immediately following the attack, but dust and debris preclude assessment of road damage. One approach to this situation would be to assess bomb damage manually to the degree possible using sources available, then update the database directly. Alternately, we can add a feature entitled "bombing event" to the database. The insertion of a bombing event triggers certain defined changes to the features it intersects. An analyst can then edit the automatically generated updates if they contradict information that is obtainable from source documents. The goal is to speed the processing of bomb damage assessment data, and to store information on bombing events in a way that supports useful analysis.

A bombing event feature is described via areal extent, temporal duration, and any helpful attributes that may assist in damage assessment and later analysis. As an example, roads that intersect a bombing event feature might be changed as follows.

- Certainty values on attributes are lowered.
- Where roads intersect the center of the bombed area, the "EXS" value is changed to "nonoperational."
- Where roads intersect the edges of the bombed area, the "RST" value is changed to "fair weather."

Figure 12 illustrates this strategy. The illustrated bombing event has a different impact on roads depending on distance from center. Note that changes resulting from the bombing event would still permit a shortest-path query to reach Point B from Point A if low certainties were tolerable and fair-weather travel possible. An analyst would review the automated values generated from the bombing event's entry into the database and adjust certainties, geometry, or attribute values, as needed. The bombing event feature would remain in the database, timestamped and available for later queries.

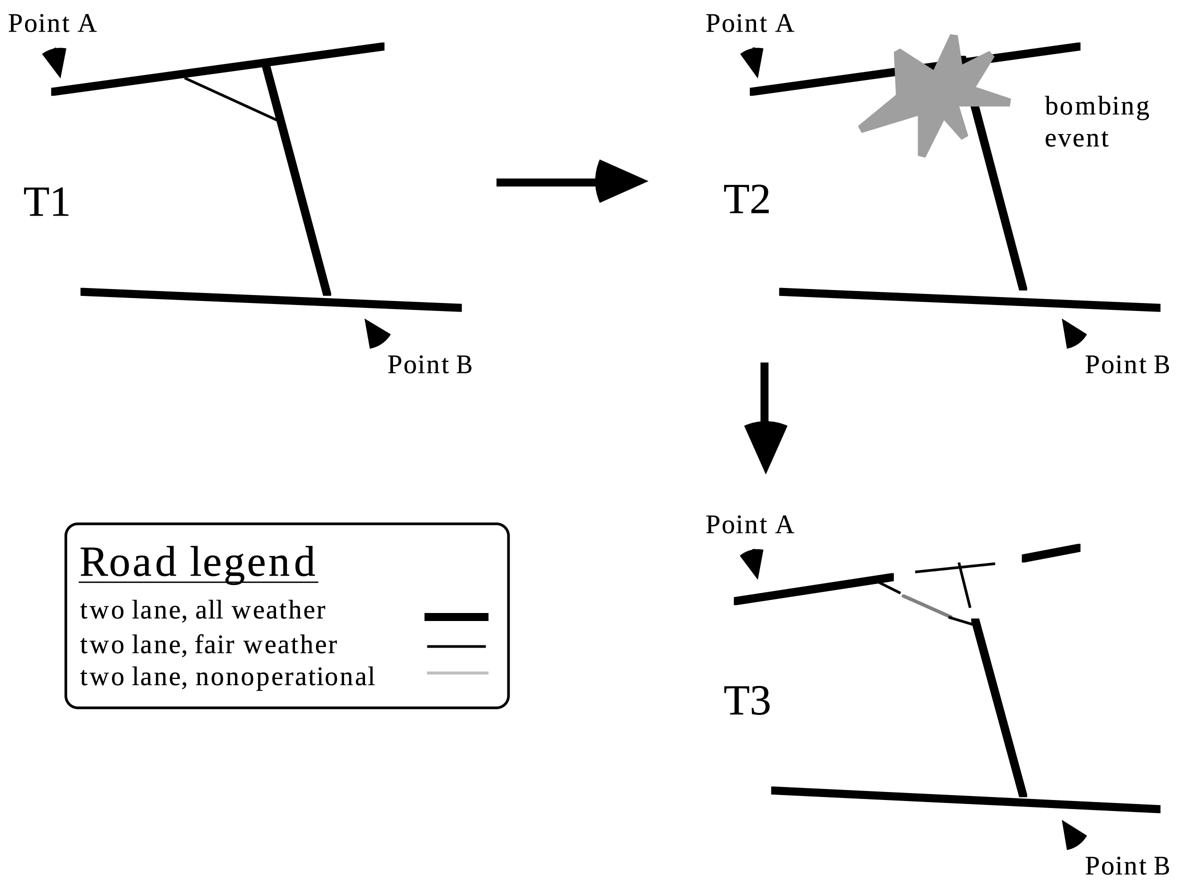


Figure 12. At T1, a vehicle can easily pass from Point A to Point B in any weather. Following the bombing event at T2, the route still is passible but only during fair weather.

The advantages of this approach are as follows.

- It is possible to analyze data with more complete information.
- It is possible to query specifically for bombing events within given areas, timeframes, and parameters.
- It is possible to examine how rapidly lines of communication were repaired following previous bombing events by referring to the past-tense database. This information could be used to extrapolate how quickly a recently bombed roadway or airfield will be back in service.

It may be useful to examine each of the above advantages more completely. Certainly network analysis can occur if terrain features ("state data") are updated directly, without resorting to digitizing event data. However, lacking good reconnaissance on bomb damage, we could still hypothesize severity of bomb damage, and even entertain various hypotheses. One approach would be to change model values to hypothesize greater or lesser destruction at different radial distances from the center, roll back the database to retrieve the pre-bombed state, and re-apply the bombing event with new model parameters. This type of capability, which could be automated, provides for more flexible "what if" analysis than an environment that stores only probabilities of correctness. Similar arguments exist for treating severe weather and other dramatic natural occurrences as event features.

A second advantage of processing events as features is it permits us to query them directly. Then, changes to terrain features that resulted from events can be re-assessed in light of later evidence. For example, if it were discovered that tricks had been employed systematically to make bomb damage appear more or less extensive than the actuality, we might wish to recall all bombing event features and re-examine how the database changed as a result. This could help to remove errors that persist due to misplaced beliefs. In addition, event data can help in assessing new evidence: rubble remaining from bombing events could be mistaken as construction debris (Figure 13), or standing water after heavy rainfall could be classed erroneously as hydrology.

A third advantage of maintaining event features is to predict future behavior based on the past. Cumulative data that describe speed of road repair provide valuable information about ability to mobilize resources, and capability of resources. Data describing absorption of standing water following heavy rains provide information on soil permeability that may not otherwise be available. While it would be unwise to draw firm conclusions based on one event, a series of events paired with subsequent state changes can strongly indicate future effects of events on terrain feature changes.

The planned prototype will implement a limited number of event features for demonstration purposes. To implement event features requires defining them within the framework of FACC, and adding methods to cascade changes through the database based on attribute values and feature intersections.

12. Summary of this work to date

This report summarizes the conceptual work that precedes a prototyping effort. It discusses critical issues of temporal data management and usage, and identifies a set of timestamps and versioning procedures that are the building blocks of spatiotemporal data management.

Many issues remain open because their approach is application-dependent. For those issues, the report has identified alternatives and described plusses and minuses. Open issues include built-in queries (e.g., to exploit relationships between world and database time), error correction, details of geometric and topological modeling, integrity checks, and extensions to the feature-attribute breakdown that govern representation of events.

A one-size-fits-all system design would likely be a poor fit for many; therefore, the goal of this continuing effort is to define the fundamental design decisions that must be addressed by any spatiotemporal system, and various design alternatives. The prototype will illustrate one approach to meeting a specific set of requirements and provide information on performance and architectural issues.

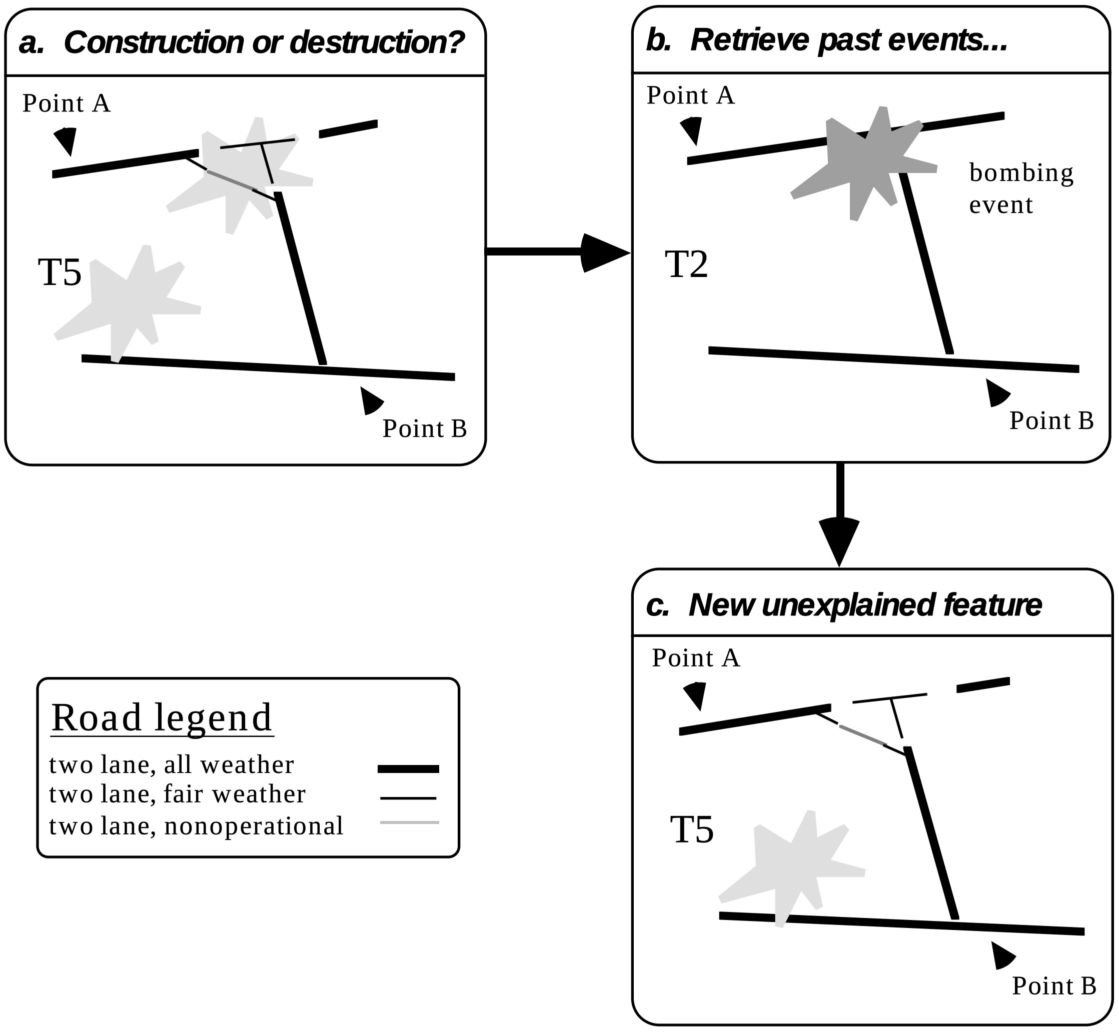


Figure 13. Resolving questions about new data. a) New source shows two areas of disturbance, either of which could reflect construction activity, possibly to add to the road network. b) By examining the database, we see that one area could be explained by historic bomb debris. c) The other area is unexplained and must be considered a new feature to be identified.